Source-to-Source Transformations for WCET Analysis: The CoSTA Approach

Adrian Prantl[*]

TU Vienna, Austria
adrian@complang.tuwien.ac.at

Abstract. *Worst-case execution time* (WCET) analysis is concerned with computing upper bounds of the maximum computation time of a program. This is indispensable for the development of safety-critical real-time systems, where missing a deadline can have disastrous consequences, including the loss of lives. Tools for WCET analysis typically analyze the object-code of a program since this is the code which is actually executed. Simultaneously, they usually rely on user-provided *annotations* such as loop-bounds or execution frequencies of program statements in order to be effective. From the perspective of a programmer, it is often more adequate to provide such information on the source code level than on the object code level. This, however, introduces a gap between the WCET annotation and the WCET analysis level. Within the CoSTA project (*Compiler Support for Timing Analysis*) we are aiming at bridging this gap. Fundamental to this is to provide appropriate new compiler support allowing to transform source code annotations into equivalent object code annotations.
In this paper we outline the approach taken in the CoSTA project to achieve this. In this project, which has recently been started, the compilation process is decomposed into a high-level machine-independent and a low-level machine-dependent two-stage process. Here, we will focus on the first stage of this process, the *high-level source-to-source compiler* and the *annotation framework*.

1 Background and Motivation

For safety-critical real-time systems the timing behavior is as important as the correctness of the calculations, since the consequences of missing a deadline can be equally catastrophic as an incorrect calculation, causing even the loss of lives. Before deploying such a system it is thus indispensable to ensure that the system meets in addition to its functional constraints also its timing constraints. Determining the *worst-case execution time* (WCET) of a program as precisely as possible is essential for this.

Intuitively, the determination of the WCET of a program is equivalent to the search for the most time-consuming path in the control flow graph of the

[*] This work has been supported by the Austrian Science Fund (Fonds zur Förderung der wissenschaftlichen Forschung) under contract P18925-N13.

program. An early approach for WCET analysis is called *timing schema* [10]. In this approach, the execution time of each basic block is assumed to be a constant and the number of iterations of each loop construct to be bounded by an upper limit, while branches are replaced by the $max()$-function.

A more sophisticated approach for supplying path information to the WCET calculation tool is based on *linear flow constraints* [11]. In this approach the program flow information - often called *flow facts* - is expressed as a system of inequalities that forms the input of an *integer linear programming* (*ILP*) problem that can be solved efficiently by a variety of tools [8]. This method is also called *implicit path enumeration technique* (*IPET*). It is implemented by commercially available tools like AiT [2] and Bound-T [4].

While it is often possible to automatically extract flow facts from the program code, it is usually necessary to require the programmer to (additionally) *manually* annotate the program with appropriate flow facts. On the one hand, this is necessary because the overall problem is undecidable (such as the determination of loop bounds). On the other hand, the programmer might have additional knowledge about the input data.

State-of-the-art tools perform the WCET-calculation on the *object code level*, which is as close as possible to the code that will eventually run on the target hardware. These tools expect that any user annotations are provided in the object code. This, however, is very demanding for a programmer and error-prone. Moreover, it implies to reassure the correctness of the annotations after each compiler run during the development phase.

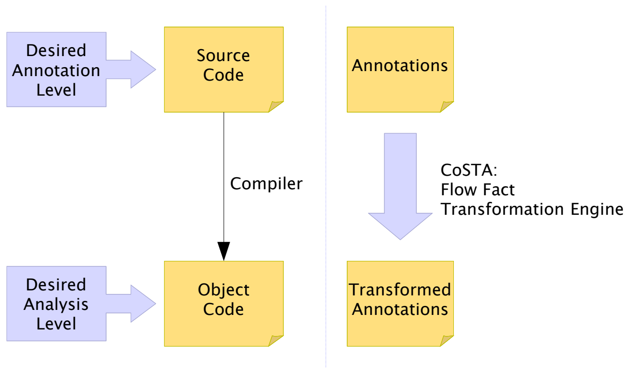

Fig. 1. Bridging the gap between annotation and analysis level

The CoSTA project aims at improving this situation. By developing and providing suitable compiler support it aims at allowing the user to add the annotations to the source code of a program which are then – together with the source code – transformed to the object code level when compiling the program.

In this paper we present and discuss the overall architecture of the system we develop in the CoSTA project, highlight the essential benefits envisioned, and discuss important features of the current state of the prototype implementation. In particular, we highlight the important role of optimizing source-to-source transformations for the overall approach. They are crucial for generating high-performance object code and for ensuring portability especially of the first stage of our approach.

2 The CoSTA-Architecture for Source-based Annotations

As indicated in the previous section, the CoSTA project strives for bridging the gap between source code annotations and object code WCET calculation. More specifically, this shall be achieved and demonstrated by developing and implementing a safe transformation framework for flow facts [5], where we target a subset of the C++ language as the programming language. The final framework shall seamlessly interact with existing IPET-based calculation tools.

As discussed before, we expect that such a source-based WCET-annotation framework makes WCET analysis more easily amenable to a programmer and overall more effective. In particular, we perceive the following benefits to be of particular value.

Validation. Automatically computed annotations on the source code level can easily be verified by the user. The increased trust in the reliability of such an analysis tool should help to reduce the amount of annotations a user makes manually.

Refinement. The user can introduce his domain-specific knowledge to provide additional information which is beyond the scope of the automatic analysis, and which can then be integrated into a cyclic work flow of perpetual refinement.

Visualization. With applying source-to-source transformations, the user can conveniently follow the steps of the compiler and thus fine-tune optimization options according to their impact on the WCET.

Figure 2 illustrates the architecture of the CoSTA approach to achieve these goals. Fundamental is the decomposition of the system into a *high-level source-to-source transformation framework* (first stage) and a *low-level WCET-aware code generation back end* (second stage).

This decomposition is motivated by the fact that many optimizations can be performed at a very high abstraction level; in our case the *abstract syntax tree* (*AST*). This way, the optimization step is independent from the target machine tool chain, but may still be parameterized to reflect specific machine characteristics. Moreover, if the WCET-critical optimizations can be moved to the source code level, it suffices to employ a relatively simple back-end compiler to finally transform the (optimized) source code into assembly language. We define *WCET-critical optimizations* as optimizations that change the control flow, thus invalidating any annotations (which are in turn assertions about the control flow graph (CFG)). An example of a CFG-modifying optimization is *loop unrolling*, which directly modifies the iteration count of a loop.

First stage. The prototype implementation of the first stage of our system uses the SATIrE[1] framework which is being developed by Markus Schordan at TU

[1] `http://www.complang.tuwien.ac.at/markus/satire/`

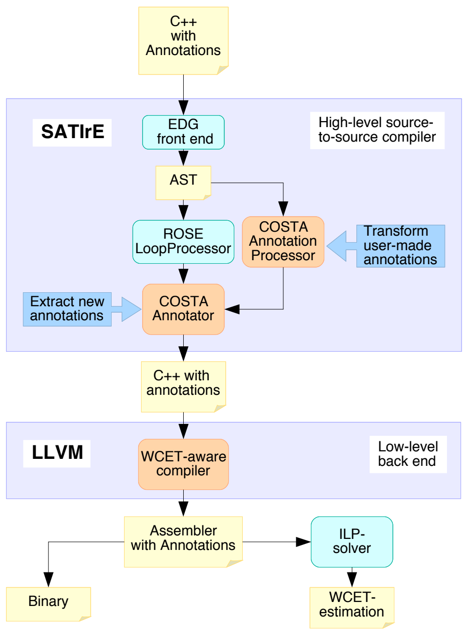

Fig. 2. A schematic overview of the CoSTA architecture

Vienna [13]. Intuitively, SATIrE is a tool environment that integrates the high-level source-to-source transformation and program analysis framework LLNL-ROSE[2] [14] with other program analysis tools, such as the *Program Analysis Generator* (*PAG*) from AbsInt [9]. For the purpose of the CoSTA project it is particularly important that SATIrE provides an external representation of the abstract syntax tree (AST) of a C++ program which can be both written to and read from. Moreover, this representation uses a syntax which can be interpreted as terms of the Prolog language. This allows us to specify program transformations directly in the Prolog language, using predicates to implement a term rewriting system. In fact, this approach was chosen to implement the first stage of our prototype.

The LLNL-ROSE framework contains a sophisticated loop optimization tool which has its roots in the Fortran D compiler. The tool can handle generic C++ programs and outputs C++ code that is very close to the original input; even templates are preserved. In the current CoSTA implementation, we use this tool to gain access to high-level optimization functions.

Second stage. The second stage of our system, the code generation back end is currently being implemented on the basis of LLVM[3], a relatively new compiler infrastructure based on a low-level virtual machine and SSA graphs that

[2] `http://www.llnl.gov/CASC/rose/`
[3] `http//www.llvm.org/`

Original user-annotated program

```
int f(int a[]) {
  for(int i=0; i<N; i+=1) {
    if (a[i] < 0) {
      // domain-spec. knowledge
      Restriction M1 <= 24
      Marker M1;
      ...
} } }
```

After loop unrolling with factor 2

```
int f(int a[]) {
  for(int i=0; i<N; i+=2) {
    if (a[i] < 0) {
      Restriction M1 <= 24/2
      Marker M1;
      ...
    }
    if (a[i] < 0) {
      Restriction M2 <= 24/2
      Marker M2;
      ...
} } }
```

Fig. 3. Transformation of user-specified annotations

is implemented in C++ [7]. We plan to implement a WCET-aware instruction selection mechanism for complex hardware architectures on this basis. The back end will only use optimizations that are not WCET-critical. This means, they will not further modify the control flow graph of the program.

In the course of implementing a safe flow facts transformation framework, the key component is the *CoSTA Annotation Processor* which is currently under implementation. The Annotation Processor takes a user-annotated program and a sequence of optimizations as input and transforms the annotations according to a set of rules. It then inserts the updated annotations into the optimized program source. Figure 3 shows an example of such a transformation.

In the following section we highlight the key components of the first stage of our system, the CoSTA Annotator and the CoSTA Annotation Processor.

3 The CoSTA Annotator and the CoSTA Annotation Processor: Extracting and Transforming Flow Facts

The *CoSTA Annotator* automatically extracts flow facts information of a program. The CoSTA Annotator thus offers an alternate route to obtain annotated source code. In particular, it avoids bothering a user to manually annotate control flow information which can be automatically extracted from the source code of the program. In fact, in many cases flow facts like loop bounds can be automatically found by a *static analysis* of the program. On the other hand, flow facts might depend on domain-specific knowledge about input-data which is usually beyond the scope of static analyses. It is thus worth noting that the CoSTA Annotator is orthogonal to the *CoSTA Annotation Processor*. The latter transforms annotations alongside optimizing transformations it applies to a program. Exemplary, we will now discuss the automatic bounding of loops:

The automatic finding of upper bounds of loop constructs is one of the tasks of the CoSTA Annotator. Currently, the algorithm operates on counter-based *for*-loops. Since C and C++ do not have a strict *for*-statement in the sense of Fortran or Pascal, loops have to satisfy a few extra conditions to be analyzable. These conditions are verified by the Annotator in advance: Each *for*-statement consists of initializer, condition, increment and body. The loop has to be induction variable based, i.e., initializer, condition and increment have to modify and test the same variable. The loop body may not contain a write access to the induction variable. The initializer may be empty. The loop may not contain early exits, such as a *break* or *return* statement. To give the programmer a little more flexibility, we provide a preprocessor that transforms *while*-loops into for loops in case they satisfy these very conditions. Using the SATIrE-framework, we were able to implement this preprocessor in very few lines of Prolog.

The implemented loop-bounding algorithm uses two strategies of varying precision. The first approach uses *symbolic evaluation* of terms in the source code to solve the equation $Bound = (End - Start)/Step$ for the induction variable i. In order to solve the equation, the terms are transformed according to a set of *rules* that exploit algebraic properties like commutativity to reduce the equation term to a single value (Figure 5). In order to reach a fixpoint and thus to guarantee termination, the rules have to satisfy a *monotonicity* property. Here, this means that the term has to shrink with each application of a rule. It should be noted that this rule-based equation solver is a good example illustrating the benefits resulting from using Prolog as implementation language. To extract the necessary information about the possible values of the program variables, every node in the AST is decorated with a pre- and a postcondition which hold the possible values of all integer variables. The loop bounds that are shown in the listing were derived by the symbolic analysis. Often the value of a variable is

```
 1  // {empty}
 2  for (int i = 0; i < 42; i += 8) {
 3    // {i_range ∈ [0..41]}
 4    // LoopBound = 6
 5    for (int j = i; j < min(42, i+8); j += 1) {
 6      // {j_range ∈ [0..41], j_symbolic ∈ [i..i + 8]}
 7      // LoopBound = 8
 8    }
 9    // {j = 42}
10  }
11   // {i = 48}
```

Fig. 4. Analysis information for loop bounds

known to be within a certain *range*, as it is the case with the induction variable in the body of a loop. This range information is important for the second strategy used by the analysis, which is intended as a fallback in case the first one fails to find a precise result. Using the range information, it is often still possible to provide a conservative estimate for many loops. As can be seen in Figure 4, the analysis information gathered by the two strategies is of varying precision. While the *range* information tends to be more pessimistic, it can unfold its whole potential when it is combined with the equation solver. The major advantage of the symbolic approach is its ability to cancel out subterms of the equation such as i in line 6 of the example in Figure 4.

A More Complex Example

We conclude this section with discussing a more complex example, which is displayed in Figure 6. The original source code of the program consists of three nested loops that perform a multiplication of two two-dimensional arrays and

```
% (a+b)-b = a
transformation(sg_subtract_op(sg_add_op(E1, E2, _, _), E3, _, _),
               E1) :-
  term_identical(E2, E3).
% (v+i1)-i2 = v+i'
transformation(sg_subtract_op(sg_add_op(E1, E2, _, _), E3, _, _),
               sg_add_op(E1, E4, _, _)) :-
  isIntVal(E2, X), isIntVal(E3, Y), Z is X-Y, isIntVal(E4, Z).
...
```

Fig. 5. Excerpt from the rule base

accumulate the result into a third array. This source code is now processed by the ROSE loop optimizer (Figure 7). First *Loop Blocking* is performed, using a block size of 8, which should improve the locality of the memory accesses, then, the innermost loop is also being unrolled by a factor of 2. This necessitates an extra loop to be created to take care of the last element in case the total number of iterations is odd. Note that ROSE instantiates the uses of the macro N, which is important for the following step: Compared with the original loop, the resulting loop conditions are quite complex, but due to their constant bounds, they are fully analyzable. The Annotator can now traverse the AST top-down and use the existing information to solve the bounding equations for each for-loop (Figure 8). Consider the induction variables of the newly generated outer loops; for the loop bodies, only a range can be found by the analysis. However, this does not affect the precision of the analysis, since their occurrences in initializer and condition of the inner loops cancel each other out. For the newly added remainder part of the innermost loop, the initializer is missing. In this case the analysis has to use the postcondition of the previous for-statement to know about possible start values for k.

```
#define N 50
int i,j,k;
double a[N][N], b[N][N], c[N][N];
...
for (i = 0; i <= N-1; i+=1) {
  for (j = 0; j <= N-1; j+=1) {
    for (k = 0; k <= N-1; k+=1) {
      c[i][j] = c[i][j] + a[i][k] * b[k][j];
} } }
```

Fig. 6. Phase 1: Original Code

4 Additional Benefits and Outlook

In this section we highlight some additional benefits of our overall approach and provide an outlook to further extensions.

It is worth noting that the framework presented in the previous sections offers two alternatives to create correctly annotated optimized code. The first one is to manually annotate the program, and then to optimize the annotated program. In this case the CoSTA Annotation Processor will update the annotations alongside the program optimization transformations. The second one is to first optimize the program and then to use the CoSTA Annotator to automatically annotate the optimized program. Figure 9 illustrates both alternatives. This flexibility

```
int _var_2;  int _var_1;  int _var_0;
int i;  int j;  int k;
double a[50][50];  double b[50][50];  double c[50][50];
...
for (_var_1 = 0; _var_1 <= 49; _var_1 += 8) {
  for (_var_2 = 0; _var_2 <= 49; _var_2 += 8) {
    for (_var_0 = 0; _var_0 <= 49; _var_0 += 8) {
      for (i = _var_2; i <= min2(49,_var_2 + 7); i += 1) {
        for (j = _var_1; j <= min2(49,_var_1 + 7); j += 1) {
          for (k = _var_0; k <= -1 + min2(49,7 + _var_0); k += 2) {
            (c[i])[j] = (((c[i])[j]) + (((a[i])[k]) * ((b[k])[j])));
            (c[i])[j] = (((c[i])[j]) + (((a[i])[1 + k]) * ((b[1 + k])[j])));
          }
          for (; k <= min2(49,7 + _var_0); k += 1) {
            (c[i])[j] = (((c[i])[j]) + (((a[i])[k]) * ((b[k])[j])));
} } } } } }
```

Fig. 7. Phase 2: Cache optimized code (Loop Blocking + Unrolling)

```
...
for (_var_1 = 0; _var_1 <= 49; _var_1 += 8) {
  #pragma WCET_LOOP_BOUND 7
  for (_var_2 = 0; _var_2 <= 49; _var_2 += 8) {
    #pragma WCET_LOOP_BOUND 7
    for (_var_0 = 0; _var_0 <= 49; _var_0 += 8) {
      #pragma WCET_LOOP_BOUND 7
      for (i = _var_2; i <= min2(49,_var_2 + 7); i += 1) {
        #pragma WCET_LOOP_BOUND 8
        for (j = _var_1; j <= min2(49,_var_1 + 7); j += 1) {
          #pragma WCET_LOOP_BOUND 8
          for (k = _var_0; k <= (-1) + min2(49,7 + _var_0); k += 2) {
            #pragma WCET_LOOP_BOUND 5
            c[i][j] = c[i][j] + a[i][k] * b[k][j];
            c[i][j] = c[i][j] + a[i][1 + k] * b[1 + k][j];
          }
          for (; k <= min2(49,7 + _var_0); k += 1) {
            #pragma WCET_LOOP_BOUND 2
            c[i][j] = c[i][j] + a[i][k] * b[k][j];
} } } } } }
```

Fig. 8. Phase 3: Automatically annotated optimized code

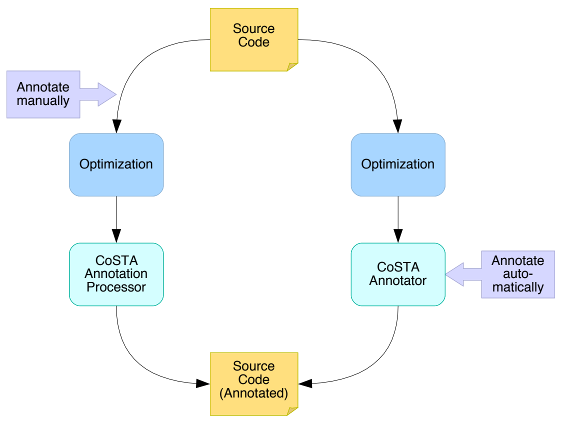

Fig. 9. Two routes to optimized annotated source code are provided in CoSTA

and dualism is only possible by using a high-level optimization approach as in our framework. Note, however, that the high-level approach requires a compiler back-end that guarantees to preserve the control flow in a way that does not alter the worst-case timing behavior of the program. Constructing such a back end is work-in-progress as part of the CoSTA project.

On modern processors, hardware resource allocation conflicts can trigger *timing anomalies*, where a locally faster execution increases the total execution time [16, 12]. Thus, another focus of our work on the compiler back end is to research scheduling and instruction selection algorithms for increased predictability.

In complex hardware architectures using features such as pipelines or instruction and data caches, the timing of an instruction depends highly on the execution history. Currently, the majority of annotation languages do not allow to specify explicit execution paths [6]. If these features shall be considered by a WCET calculation, it will be necessary to undertake a deeper look at annotation languages. While there are already approaches to integrate execution context information into the IPET calculation method [1], it will be necessary to create adequate annotation methods for context sensitive path descriptions as well.

5 Conclusions

The CoSTA project aims at making WCET analysis more effective and more amenable, especially by lifting the annotation level from the object code level to the source code level. Experiences with the current prototype implementation indicate that the chosen system architecture is well-suited to meet these goals. In particular, our experiences with optimizing source-to-source transformations indicate that these benefits can be achieved without sacrificing the performance of the object code of the application programs. Currently, we work on integrating more refined algorithms for automatic flow facts extraction and further source-to-source transformations. Simultaneously, we work on connecting the high-level first stage of our system with its low-level second stage, which will enable us to link our system to existing WCET analysis tools. As another strand of research in the CoSTA project we consider the development of advanced annotation languages which are even more suitable to reach the goals of the CoSTA project. In fact, investigating the adequacy of the commonly used annotation languages for this purpose, it turned out that all these languages have their own strengths and limitations motivating us to rise the *annotation language challenge* [6] – a challenge being closely related to and complementing the previously launched WCET tool challenge [3, 15].

Acknowledgements. The author would like to thank the members of the CoSTA-project team for discussions related to the subject of this paper. In particular, he would like to thank Jens Knoop for helpful comments on earlier versions of this paper.

References

1. J. Engblom and A. Ermedahl. Modeling complex flows for worst-case execution time analysis. In *Proceedings 21st IEEE Real-Time Systems Symposium (RTSS)*, Orlando, Florida, USA, Dec. 2000.
2. C. Ferdinand. Worst case execution time prediction by static program analysis. *18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*, 03:125a, 2004.
3. J. Gustafson. The WCET tool challenge 2006. In *Preliminary Proceedings 2nd Int. IEEE Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, pages 248 – 249, Paphos, Cyprus, November 2006.
4. N. Holsti and S. Saarinen. Status of the Bound-T WCET tool. In *Proceedings Euromicro Worst-Case Execution Time Workshop 2002 (WCET 2002)*, 2003.
5. R. Kirner. *Extending Optimising Compilation to Support Worst-Case Execution Time Analysis*. PhD thesis, Technische Universität Wien, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, May 2003.
6. R. Kirner, J. Knoop, A. Prantl, M. Schordan, and I. Wenzel. WCET Analysis: The Annotation Language Challenge. In *Proceedings 7th Int'l Workshop on Worst-Case Execution Time (WCET) Analysis*, 2007. To appear.
7. C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.
8. Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *Proceedings 32nd ACM/IEEE Design Automation Conference*, pages 456–461, June 1995.
9. F. Martin. PAG – an efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer*, 2(1):46–67, 1998.
10. C. Y. Park and A. C. Shaw. Experiments with a program timing tool based on a source-level timing schema. *Computer*, 24(5):48–57, May 1991.
11. P. Puschner and A. V. Schedl. Computing maximum task execution times – a graph-based approach. *Journal of Real-Time Systems*, 13:67–91, 1997.
12. J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker. A definition and classification of timing anomalies. In F. Mueller, editor, *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.
13. M. Schordan. Combining tools and languages for static analysis and optimization of high-level abstractions. In *Proceedings 24. Workshop of "GI-Fachgruppe Programmiersprachen und Rechenkonzepte"*, 2007.
14. M. Schordan and D. Quinlan. Specifying transformation sequences as computation on program fragments with an abstract attribute grammar. In *Proceedings Fifth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'05)*, pages 97–106. IEEE Computer Society Press, 2005.
15. L. Tan and K. Echtle. The WCET tool challenge 2006: External evaluation – draft report. In *Handout at the 2nd Int. IEEE Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, Paphos, Cyprus, November 2006. 13 pages.
16. I. Wenzel, R. Kirner, P. Puschner, and B. Rieder. Principles of timing anomalies in superscalar processors. In *Proceedings 5th International Conference on Quality Software*, Sep. 2005.